Reuse-based Optimization for Pig Latin

Jesús Camacho-Rodríguez, Dario Colazzo, Melanie Herschel, Ioana Manolescu, Soudip Roy Chowdhury

Reuse-based Optimization for Pig Latin

Jesús Camacho-Rodríguez
Université Paris-Sud & Inria
France
jesus.camacho-rodriguez@lri.fr

Dario Colazzo
Université Paris-Dauphine
France
dario.colazzo@dauphine.fr

Melanie Herschel
Université Paris-Sud & Inria
France
melanie.herschel@lri.fr

Ioana Manolescu
Inria & Université Paris-Sud
France
ioana.manolescu@inria.fr

Soudip Roy Chowdhury
Inria & Université Paris-Sud
France
soudip.roy-chowdhury@inria.fr

ABSTRACT

Pig Latin has become a popular language within the data management community interested in the efficient parallel processing of large data volumes. The dataflow-style primitives of Pig Latin provide an intuitive way for users to write complex analytical queries, which are in turn compiled into MapReduce jobs.

Currently, subexpressions occurring repeatedly in Pig Latin scripts are executed as many times as they occur, leading to avoidable MapReduce jobs. The current Pig Latin optimizer is not capable of recognizing, and thus optimizing, such repeated subexpressions.

We present a novel approach for identifying and reusing common subexpressions occurring in Pig Latin scripts. In particular, we lay the foundation of our reuse-based algorithms by formalizing the semantics of the Pig Latin query language with extended nested relational algebra for bags. Our algorithm, named *PigReuse*, operates on the algebraic representations of Pig Latin scripts, identifies subexpression merging opportunities, selects the best ones to execute based on a cost function, and merges other equivalent expressions to share its result. Our experimental results demonstrate the efficiency and effectiveness of our reuse-based algorithms and optimization strategies.

1. INTRODUCTION

The efficient processing of very large volumes of data has lately relied on massively parallel processing models, of which MapReduce is the most widely adopted. However, the simplicity of the MapReduce model leads to relatively complex programs to express even moderately complex tasks. To facilitate the specification of data processing tasks to be executed in a massively parallel fashion, several higher-level query languages have been introduced, which are more user-friendly, and which are automatically compiled into MapReduce programs. Languages that have gained wide adoption include Pig Latin [21], HiveQL [29], or Jaql [4].

In this work, we consider the Pig Latin language, which has raised significant interest from the application developers as well as the research community. Pig Latin provides dataflow-style primitives for expressing complex analytical data processing tasks. Pig Latin programs (also named *scripts*) are automatically optimized and compiled into MapReduce jobs by the Apache Pig system [22].

In a typical batch of Pig Latin scripts, there may be many repeated sub-expressions, that is: script fragments applying the same processing on the same inputs, but appearing in distinct places within the same (or several) scripts. While the Pig Latin engine includes a query optimizer, it is currently not capable of recognizing such repeated subexpressions. As a consequence, they will be executed as many times as they appear in the Pig Latin script batch, whereas there is obviously an opportunity for enhancing performance by identifying common subexpressions, executing them only once, and reusing the results of the computation in every script needing them.

Identifying and reusing common subexpressions occurring in Pig Latin scripts automatically is the target of the present work. The problem bears obvious similarities with the known multi-query optimization and workflow reuse problems; however, as we discuss in Section 6, the Pig Latin setting leads to several novel aspects of the problem, which lead us to propose dedicated algorithms to solve them.

Motivating example. A Pig Latin script consists of a set of *binding expressions* and *store expressions*. Each binding expression follows the syntax var = op, meaning that the expression op will be evaluated, and the bag of tuples thus generated will be bound to the variable var. Then, var can be used by follow-up expressions in a script.

Consider the following Pig Latin script a_1:

```
1 A = LOAD 'page_views' AS (user, time, www);
2 B = LOAD 'users' AS (name, zip);
3 C = JOIN A BY user, B BY name;
4 D = FOREACH C GENERATE user, time, zip;
5 STORE D INTO 'a1out1';
6 E = JOIN A BY user LEFT, B BY name;
7 STORE E INTO 'a1out2';
```

Line 1 loads data from a file `page_views` and creates a bag of tuples that is bound to variable `A`. Each of these tuples consists of three attributes `(user,time,www)`; the schema is specified dynamically. Line 2 loads data from a second file, and binds the resulting tuple bag to `B`. Line 3 joins the tuples of `A` and `B` based on the equality of the values bound

to attributes `user` and `name`. The next line uses the important Pig Latin operator `FOREACH`, that applies a function on every tuple of the input bag. In this case, line 4 projects the attributes `user`, `time` and `zip` of every tuple in `C`. Then the result is stored in the file `a1out1`. In turn, line 6 executes a left outer join over the tuples of `A` and `B` based on the equality of the values bound to the same attributes `user` and `name`, and the result is stored in `a1out2`.

The following script a_2 only executes a left outer join over the same inputs:

```
1 A = LOAD 'page_views' AS (user, time, www);
2 B = LOAD 'users' AS (name, zip);
3 C = JOIN A BY user LEFT, B BY name;
4 STORE D INTO 'a2out';
```

The script b that we introduce next produces the same outputs as a_1 and a_2:

```
1 A = LOAD 'page_views' AS (user, time, www);
2 B = LOAD 'users' AS (name, zip);
3 C = COGROUP A BY user, B BY name;
4 D = FOREACH C GENERATE flatten(A), flatten(B);
5 E = FOREACH D GENERATE user, time, zip;
6 STORE E INTO 'a1out1';
7 F = FOREACH C GENERATE flatten(A),
8               flatten (isEmpty(B) ? {(null,null,null)} : B);
9 STORE F INTO 'a1out2';
10 STORE F INTO 'a2out';
```

However, *b's execution time is* 45% *of* a_1 *and* a_2 *times combined*. The reason is twofold. First, observe that the joins are rewritten into a `COGROUP`[1] operation (line 3) and `FOREACH` operations (lines 4 and 7-8). The interest of cogroup is that through some simple restructuring, one can carve out of the cogroup output various flavors of joins (natural, outer, nested, semi etc.) This restructuring operation differs depending on whether we want to generate the join between `A` and `B` needed for script a_1 (line 4), or the left outer join between `A` and `B` for scripts a_1 and a_2 (lines 7-8). The detailed semantics of these restructuring operations will become clear in Section 4. Thus, the first reason for the speedup is that the `COGROUP` output is reused to generate the result for both joins. In turn, the second reason is that the left outer join is computed only once, and then the result is output for scripts a_1 (line 9) and a_2 (line 10).

Contributions. The technical contributions of this work are the following.

- We formalize the representation of Pig Latin scripts based on an existing well-established algebraic formalism, specifically Nested Relational Algebra for Bags (NRAB) [10]. This provides a formal foundation for accurately identifying common expressions in batches of Pig Latin scripts.
- We propose PigReuse, a multi-query optimization algorithm that merges equivalent subexpressions it identifies in Directed Acyclic Graph (DAGs) of NRAB operators corresponding to PigLatin scripts. After identifying such reutilization opportunities, PigReuse produces an optimal merged plan where redundant computations have been eliminated. PigReuse relies on an efficient Binary Integer Linear Programming (BIP, in short) solver to select the best plan based on the cost function provided.
- We present extensions to our baseline PigReuse optimization algorithm to improve its *effectiveness*, i.e., increase the number of common subexpressions it detects.
- We have implemented PigReuse as an extension module for the Pig system. We present an experimental evaluation of our techniques using two different cost functions to select the best plan.

[1] `COGROUP` can be seen as a generalization of the *group-by* operation on two or more relations: for every value of the grouping key occurring in any of the inputs, it outputs a tuple that includes an attribute *group* bound to the grouping key, and a bag of tuples for each input R_i such that the bag R_i includes all tuples in R_i that contain the value of the grouping key.

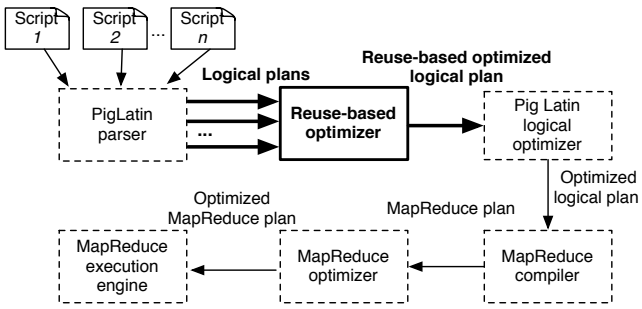


Figure 1: Integration of PigReuse optimizer within Pig Latin execution engine.

Outline. Section 2 describes our approach to represent Pig Latin scripts as DAGs of NRAB operators. Section 3 presents PigReuse, our reuse-based query optimization approach focusing on identifying and merging common subexpressions. Then, Section 4 details different strategies that we use to make our reuse-based optimization approach more effective. Section 5 describes our experimental evaluation. Finally, Section 6 discusses related work, and then we conclude.

2. ALGEBRAIC REPRESENTATION OF PIG LATIN PROGRAMS

Figure 1 depicts the integration of our reuse-based optimization into the Pig Latin architecture; modules, denoted by dashed lines, belong to the original Pig Latin query processor. As it can be seen in the figure, our reuse-based optimizer works on the algebraic representation of Pig Latin scripts. Thus, our proposal is orthogonal to the Pig Latin query evaluation and execution process. This allows our approach (*i*) to benefit from the Pig Latin optimizer, and (*ii*) to apply our optimization independently of the underlying Pig Latin query compilation and execution engines.

The algebraic formalization of Pig Latin is necessary, as it ensures the *correctness* of the manipulations involved in the detection of common subexpressions within batches of Pig Latin scripts, based on known expression equivalence results [5, 6, 19]. The Pig Latin data model features complex data types (e.g., tuple, map etc.) and nested relations with duplicates (bags). Earlier work [2] stated that Pig Latin scripts can be translated to Nested Relational Algebra with bag semantics, but to the best of our knowledge, no formalization of such translation has been proposed to date. In this section, we present our translation of PigLatin to an extension of the Nested Relational Algebra for Bags [10] (NRAB, for short) that includes operators needed to support Pig Latin semantics.

Notation	Name	Input arity	Output description
ϵ	Duplicate elimination	Unary	Distinct tuples from the input relation.
$map\langle\varphi\rangle$	Restructure	Unary	All the tuples in the input after applying a function φ.
$\sigma\langle p\rangle$	Selection	Unary	All the tuples in the input that satisfy the boolean predicate p.
$\uplus$	Additive union	n-ary, $n \geq 2$	Union of input relations, including duplicates.
$-$	Substraction	Binary	Difference between relations, including duplicates.
$\times$	Cartesian product	n-ary, $n \geq 2$	Cartesian product of input relations, including duplicates.
δ	Bag-destroy function	Unary	Unnests one level for the tuples in the input relation.

Notation	Name	Input arity	Output description
$scan\langle fileID\rangle$	Load	-	Reads a file and loads it as a relation.
$store\langle dir\rangle$	Store	Unary	Writes the contents of the tuples for an input relation to a file.
$\pi\langle a_1, \ldots, a_n\rangle$	Projection	Unary	Projects attributes $a_1, \ldots, a_n$ from the input tuples.
$cogroup\langle a_1, \ldots, a_n\rangle$	Cogroup	n-ary, $n \geq 1$	Groups tuples together from input relations based on the equality of their values for attributes $(a_1, \ldots, a_n)$.
$\bowtie\langle p\rangle$	Join	n-ary, $n \geq 2$	Returns the combination of tuples from input relations when boolean condition p over them is true.
$\mathbin{⟕}\langle p\rangle$	Left outer join	Binary	Returns the combination of tuples from input relations for which boolean condition p is true, and the tuples in the left relation without a matching right tuple.
$\mathbin{⟖}\langle p\rangle$	Right outer join	Binary	Returns the combination of tuples from input relations for which boolean condition p is true, and the tuples in the right relation without a matching left tuple.
$\mathbin{⟗}\langle p\rangle$	Full outer join	Binary	Returns the combination of tuples from input relations for which boolean condition p is true, the tuples in the left relation without a matching right tuple, and the tuples in the right relation without a matching left tuple.
$mapconcat\langle\varphi\rangle$	Restructure and concatenate	Unary	Applies $map\langle\varphi\rangle$ and concatenates its result to the original tuple.
$empty$	Empty function	Unary	The boolean function $empty$ returns true if and only if the input relation is empty.
sum, max, min, $count$	Aggregate functions	Unary	Returns the sum of integer values for an attribute field in an input relation, maximum integer value, minimum integer value of an attribute field in an input relation, and total number of tuples in an input relation.

Table 1: Basic NRAB operators (above) and proposed extension (below) to support Pig Latin semantics.

In the following, Section 2.1 provides background on the NRAB, while Section 2.2 presents our translation of Pig Latin operators into the algebra operators. Finally, Section 2.3 describes the DAG representation of the translated scripts.

2.1 Extended NRAB

We consider a subset of the NRAB algebra and extend it with other operators. Table 1 lists all basic operators of NRAB (top part) and the additional operators we introduce (bottom part). For space reasons, the formal semantics of these operators is described in Appendix A. All additional operators but *scan* and *store* are redundant, i.e., they can be expressed using the basic operators. We decided to introduce additional operators for two main reasons: (*i*) allowing a one-to-one representation of Pig Latin scripts into the algebra, and (*ii*) giving our algorithm additional opportunities to detect common subexpressions by exploring different rewritings. For instance, any type of join can also be expressed by a combination of *cogroup*, *restructure*, and *bag destroy*. Using this alternative representation of a join, it becomes easier to match it with a subexpression involving a *cogroup*, compared to searching the common subexpressions on the plans we would generate using the standard NRAB operators only.

2.2 Pig Latin Translation

Along the lines of [25], we define our Pig Latin to NRAB translation by means of deduction (or *translation*) *rules*.

$$\frac{[\![\mathsf{expr}_1]\!]_{\Gamma_0} \rightsquigarrow \Gamma_1 \quad \cdots \quad [\![\mathsf{expr}_n]\!]_{\Gamma_{n-1}} \rightsquigarrow \Gamma_n}{[\![\mathsf{expr}_1; \ldots; \mathsf{expr}_n;]\!]_{\Gamma_0} \rightsquigarrow \Gamma_n} \quad \text{(Script)}$$

$$\frac{\mathsf{op} \Rightarrow A \qquad \Gamma_1 := \Gamma_0 \cup \{var = A\}}{[\![\underline{\mathsf{var}} = \mathsf{op}]\!]_{\Gamma_0} \rightsquigarrow \Gamma_1} \quad \text{(Bind)}$$

$$\frac{A := store\langle dir\rangle(var) \qquad \Gamma_1 := \Gamma_0 \cup \{\top = A\}}{[\![\texttt{STORE } \underline{\mathsf{var}} \texttt{ INTO } \underline{\mathsf{dir}}]\!]_{\Gamma_0} \rightsquigarrow \Gamma_1} \quad \text{(Store)}$$

Figure 2: Translation rules for Pig Latin scripts and basic Pig Latin constructs.

These rules are classified in two main sets. The first one deals with the translation of programs as ordered sequences of expressions, while the second set deals with the translation of a single Pig Latin operation.

Pig Latin scripts translation. Rules in the first set are defined over judgments of the form $[\![P]\!]_\Gamma \rightsquigarrow \Gamma'$, meaning that a Pig Latin program P is translated to a set of named NRAB expressions Γ', in the context of a given set of named NRAB expressions Γ. A named NRAB expression is a binding of the form $\{\underline{\mathsf{var}} = A\}$ where $\underline{\mathsf{var}}$ is a name given to the algebraic expression A. During the application of the translation rules, every binding expression $\{\underline{\mathsf{var}} = \mathsf{op}\}$ belonging to the Pig Latin program is translated into a named algebraic expression $\{var = A\}$, where A is the NRAB ex-

$$\frac{}{\texttt{FOREACH } \underline{\text{var}} \texttt{ GENERATE } \underline{\text{var}}_1 \ldots, \underline{\text{var}}_n \Rightarrow \pi\langle \underline{\text{var}}_1, \ldots, \underline{\text{var}}_n \rangle(\underline{\text{var}})} \quad \text{(Projection ForEach)}$$

$$\frac{\underline{\text{f}}_1 \Rightarrow A'_1, \ldots, \underline{\text{f}}_m \Rightarrow A'_m \qquad A_2 := map\langle [A'_1, \ldots, A'_m] \rangle (var_1)}{\texttt{FOREACH } \underline{\text{var}} \texttt{ GENERATE } \underline{\text{f}}_1, \ldots, \underline{\text{f}}_m \Rightarrow A_2} \quad \text{(Simple ForEach)}$$

$$\frac{\begin{array}{c} \text{op}_1 \Rightarrow A'_1 \qquad A_1 := mapconcat\langle [nvar_1 = A'_1] \rangle (var_1) \\ \text{op}_i \Rightarrow A'_i \qquad A_i := mapconcat\langle [nvar_i = A'_i] \rangle (A_{i-1}) \qquad 2 \leq i \leq n \\ \underline{\text{f}}_1 \Rightarrow A''_1, \ldots, \underline{\text{f}}_m \Rightarrow A''_m \qquad A_{n+1} := map\langle [A''_1, \ldots, A''_m] \rangle (A_n) \end{array}}{\texttt{FOREACH } \underline{\text{var}}_1 \ \{\underline{\text{nvar}}_1 = \text{op}_1; \ldots; \underline{\text{nvar}}_n = \text{op}_n; \ \texttt{GENERATE } \underline{\text{f}}_1, \ldots, \underline{\text{f}}_m\} \Rightarrow A_{n+1}} \quad \text{(Complex ForEach)}$$

Figure 3: Translation rules for `foreach` operator.

pression corresponding to op (and obtained by applying the second set of translation rules).

Binding expressions in the Pig Latin program are translated one after the other, according to their order in the Pig Latin program. Each time a named algebraic expression $\{var = A\}$ is created, it is added to the context Γ. The context holds all variables which may be encountered while translating subsequent Pig Latin binding expressions of the program; we assume that *var* is a fresh variable, i.e., it is not already bound in the context.

Figure 2 shows the rules used by the high-level translation process outlined above. The rules are rather simple; note that the rule corresponding to `STORE` adds to the context a dummy binding. This rule records the fact that a bag has been saved on the disk, thus the symbol $\top$ is used instead of a variable symbol, which is not needed in this case.

Pig Latin operations translation. The second set of rules translates the operator op from a binding expression $\underline{\text{var}} = \text{op}$ into a NRAB expression A. These rules are defined over judgements of the form $\text{op} \Rightarrow A$ with the obvious meaning. Most Pig Latin operators have a one-to-one correspondence with NRAB operators, hence the related translation is straightforward (see Appendix B).

A special case is the `FOREACH` operator, whose translation is not trivial. The translation rules for this operator are shown in Figure 3. We use three different rules depending on the form of the `FOREACH` expression:

- The first rule (Projection ForEach) deals with the case of an iteration simply projecting n fields of the input relation. The rule specific to this case enables the generation of NRAB projections, playing an important role in our optimization technique. In Figure 3, $\underline{\text{var}}_1$,$\underline{\text{var}}_2$,...,$\underline{\text{var}}_n$ are the fields to be projected from the input relation denoted by the name $\underline{\text{var}}$.
- If the previous rule does not apply, and if the `FOREACH` operator contains a `GENERATE` clause with functions applied on the input relation $\underline{\text{var}}$, the second rule (Simple ForEach) is applied. In this rule, every function definition $\underline{\text{f}}_i$ inside the `GENERATE` clause is translated to an algebraic expression A'_i and these expressions are applied with a *map* operator on each tuple in $\underline{\text{var}}_1$ (recall Table 1).
- Rule (Complex ForEach) in Figure 3 considers `FOREACH` expressions containing one or more binding expressions before the `GENERATE` clause. Each Pig Latin operator op_i is translated first into an algebraic expression A'_i. These algebraic expressions are then used by a *mapconcat* operator, which applies A'_i on each tuple in A'_{i-1} (or $\underline{\text{var}}_1$ initially) and appends the result to the input tuple; the use of *mapconcat* is necessary to use local contextual information that is visible only in the scope of the translated `FOREACH` expression. Every function definition $\underline{\text{f}}_i$ inside the `GENERATE` clause is then translated to an algebraic expression A''_i and these expressions are applied on the algebraic expression A_n.

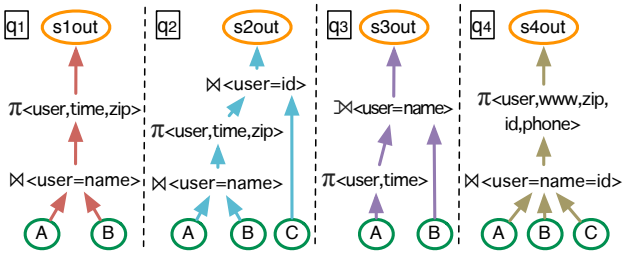
s_1
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
C = JOIN A BY user, B BY name;
D = FOREACH C GENERATE user, time, zip;
STORE D INTO 's1out';
```

s_2
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
C = LOAD 'power_users' AS (id, phone);
D = JOIN A by user, B BY name;
E = FOREACH D GENERATE user, time, zip;
F = JOIN E BY user, C by id;
STORE F INTO 's2out';
```

s_3
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
C = FOREACH A GENERATE user, time;
D = JOIN C by user LEFT, B by name;
STORE D INTO 's3out';
```

s_4
```
A = LOAD 'page_views' AS (user, time, www);
B = LOAD 'users' AS (name, zip);
C = LOAD 'power_users' AS (id, phone);
D = JOIN A BY user, B by name, C by id;
E = FOREACH D GENERATE user, www, zip, id, phone;
STORE E INTO 's4out';
```

(a)



(b)

Figure 4: Sample Pig Latin scripts (a) and their corresponding algebraic DAG representation (b).

2.3 DAG-structured NRAB queries

Let P be a Pig Latin program, and Γ be a set of NRAB binding expressions obtained from P via the translation process described above. We define below the DAG representation of Γ as follows.

Definition 1. Given a context (set of bindings) $\Gamma = \{var_1{=}A_1, \ldots, var_n{=}A_n\}$, its DAG representation is a pair $(V, \vec{E})$. Every node $v_i \in V$ is a tuple $\langle var_i, op_i^A \rangle$ such that:

- var_i is the variable associated to the node (thus, it is the unique identifier of a node);
- op_i^A is the top-most algebraic operator in the expression bound to var_i;

Every edge $e_{i,j} \in \vec{E}$ represents the data flow from node v_i to node v_j, i.e., the operation op_j^A is applied on the bag of tuples produced by op_i^A. ◇

Observe that in our DAG representation, a *source* i.e., a node with no incoming edges, always contains a *scan* operator. In turn, a *sink* i.e., a node with no outgoing edges, always corresponds to a *store* operator.

Figure 4.a introduces four different Pig Latin scripts s_1-s_4, while their corresponding algebraic representation is shown in Figure 4.b. We will reuse these sample scripts throughout this paper. The scripts read data from the three input relations `page_views`, `users`, and `power_users`; from now on, we depict these relations as `A`, `B`, and `C` in the algebraic plans, and we refer to them in the same fashion.

To illustrate, consider the script s_1, whose translation yields:

$$\begin{aligned}\Gamma = \{\ A &= scan\langle\text{`}page\_views\text{'}\rangle,\\ B &= scan\langle\text{`}users\text{'}\rangle,\\ C &= \bowtie\langle user{=}name\rangle(A,B),\\ D &= \pi\langle user, time, zip\rangle(C),\\ &store\langle\text{`}s1out\text{'}\rangle(D)\ \}\end{aligned}$$

After connecting the different algebraic expressions, we obtain the DAG query q_1 shown in Figure 4.b.

3. REUSE-BASED QUERY OPTIMIZATION

We have previously shown how to translate Pig Latin scripts into NRAB DAGs. Based on this DAG formalism, we now introduce our PigReuse algorithm that optimizes the query plans corresponding to a batch of scripts by reusing results of repeated subexpressions.

More specifically, given a collection of NRAB DAG queries Q, PigReuse proceeds in two steps:

Step (1). *Identify and merge all the equivalent subexpressions in Q.* To this end, we use an AND-OR DAG, in which an AND-node (or operator node) corresponds to an algebraic operation in Q, while an OR-node (or equivalence node) represents a set of subexpressions that generate the same result bag.

Step (2). *Find the optimal plan from the AND-OR DAG.* Based on a cost model, we make the globally best choice of the set of operator nodes that need to be evaluated. Our approach is independent of the particular cost function chosen; we discuss in Section 5.2 the functions that we have implemented for PigReuse.

The final output of PigReuse is an optimized plan that contains (*i*) the operator nodes leading to *minimizing the cummulated cost* of all the queries in Q, while *producing, together, the same set of outputs* as the original Q, and (*ii*) equivalence nodes that represent *result sharing* of an operator node with other operators in Q. In the following sections, we describe each step of our reuse-based optimization algorithm in detail.

3.1 Equivalence-based merging

To join all detected equivalent expressions in Q, we build an AND-OR DAG, which we term *equivalence graph* (EG, in short); the construction is carried out in the spirit of previous optimization works [8, 26]. In the EG, an AND-node corresponds to an algebraic operation (e.g., selection, projection etc.). An OR-node o is introduced whenever a set of expressions $e_1, e_2, \ldots, e_k$ have been identified as equivalent; in the EG, o has as children the algebraic nodes at the roots of the expressions $e_1, e_2, \ldots, e_k$. In the following, we refer to AND-nodes as *operator nodes*, and OR-nodes as *equivalence nodes*.

Formally, we define an EG as follows.

Definition 2. An equivalence graph (EG) is a DAG, defined by the pair $(O \cup A \cup T_o, E)$, such that:

- $O \cup A \cup T_o$ is the set of nodes:
  - O is the set of equivalence nodes.
  - A is the set of operator nodes.
  - T_o is the set of sink nodes.
- $E \subseteq (O \times A) \cup (A \times O) \cup (A \times T_o)$ is a set of directed edges such that:
  - Each node $a \in A$ has an indegree of at least one, and an outdegree equal to one.
  - Each node $o \in O$ has an indegree of at least one, and an outdegree of at least one.
  - Each node $t_o \in T_o$ has an indegree of at least one. ◇

Observe that in an EG, O nodes can only point to A nodes, while A nodes can only point to O nodes. In turn, T_o can only be pointed by A nodes.

An important point to stress here is that *equivalence nodes with more than one child amount to optimization opportunities* as they indicate that several operator nodes have a common (equivalent) child subexpressions. In this case, we can choose the "best" way to compute the result of the subexpression among the choices given by the OR-node. Choosing the best alternative for each such OR-node is based on a cost model, where the best plan corresponds to the plan with overall minimal cost. Optimal plan selection is discussed in detail in the next section.

Building the equivalence graph. To build the equivalence graph, we need to identify equivalent expressions within the input NRAB query set Q. We reuse the classical notion of query equivalence here, i.e., two expressions are *equivalent* iff their result is provably the same regardless of the data on which they are computed.

We build the EG in the following fashion. First, we create the EG *eg* with a single equivalence node o_s, i.e., the EG *source*. We take every NRAB query $q \in Q$ and perform a *breadth-first* traversal of its nodes. Each source node $s \in q$ is added to *eg*, and an edge (o_s, s) is created.

Subsequently, for each node n having the source node s as an input, we verify whether there exists a node n_{eg} in *eg*, such that the expression rooted in n is equivalent to the one rooted in n_{eg}.

- If such an equivalence is detected, we connect n to the equivalence node o that n_{eg} feeds.

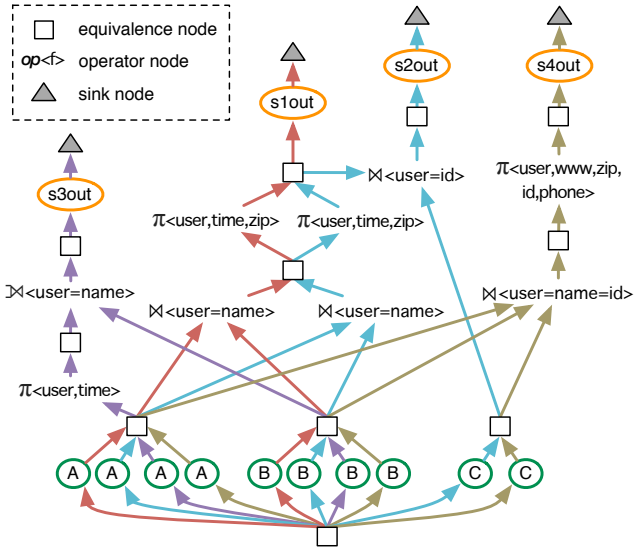


Figure 5: EG corresponding to NRAB DAGs q_1-q_4.

- If no such equivalent node is found, n is added to eg, a new equivalence node o is added to eg, and an edge (n, o) is created. In either case, for each node n' that is a parent of n in the original query, n' is added to eg and an edge (o, n') is created. Within a set of equivalent nodes, note that each node is the root of a sub-DAG that represents a NRAB expression; the expressions corresponding to all these nodes are equivalent.

To check if two expressions A, A' rooted at nodes n and n' are equivalent, we apply the known commutativity, associativity etc. laws that have been extensively studied for the bag relational algebra, as introduced in [3, 9, 24]. If one of the possible rewritings of A' (guaranteed to be equivalent to A' through the abovementioned prior work) matches the exact syntax of A, then A and A' are equivalent and they become children of the same equivalence node.

As mentioned above, the equivalent transformation rules we apply are those previously identified for NRAB, i.e., they only cover operators that have been previously defined as (extensions of) NRAB operators (i.e., $\uplus$, $-$, $\times$, ϵ, δ, *map*, σ, π, and $\bowtie$, see Table 1). As we will discuss in Section 4, we provide a set of new equivalent rewriting rules involving operators we introduced in this work (e.g., *cogroup* and outer join variants). These rules allow identifying more equivalences and thus improve over the baseline PigReuse algorithm presented in this section.

Figure 5 depicts the EG corresponding to the NRAB DAGs q_1 to q_4 in Figure 4.b. In Figure 5, we use boxes to represent equivalence nodes, while sink nodes are represented by shadowed triangles. All the leaf nodes in the NRAB DAGs that correspond to the same *scan* operation (namely, nodes A, B, and C) feed the same equivalence node. The equi-joins coming from DAGs q_1 and q_2 on relations A and B over attributes *user* and *name* are also inputs to the same equivalence node.

3.2 Cost-based plan selection

Once an EG has been generated from a set of NRAB queries, our goal is to find the best alternative plan (having the smallest possible cost) computing the same outputs (results) for any possible source instances. In other terms:

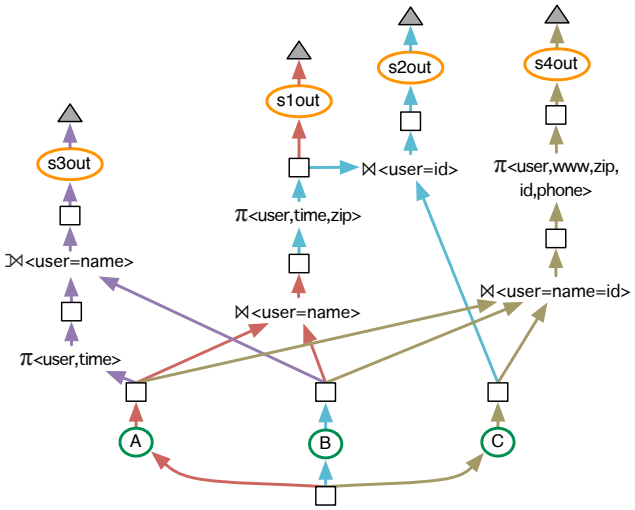


Figure 6: Possible REG for the EG in Figure 5.

we are interested in finding the minimum-cost equivalent NRAB batch.

We call the output plan a *result equivalence graph* (or REG, in short).

Definition 3. A result equivalence graph (REG) with respect to an EG defined by $(O \cup A \cup T_O, E)$ is itself a DAG, defined by the pair $(O^* \cup A^* \cup T_O, E^*)$ such that:

- $O^* \subseteq O$.
- $A^* \subseteq A$.
- The set of sink nodes T_o is identical in EG and REG.
- $E^* \subseteq E$.
- Each sink node has an indegree of exactly one.
- Each operator node indegree in the REG is equal to its indegree in the EG.
- Each equivalence node has an indegree of exactly one, and an outdegree of at least one. ◇

Clearly, the REG still produces the same outputs as the original EG, as all sink nodes are preserved. All we do is to choose exactly one among the equivalent alternatives provided by equivalence nodes. Furthermore, there is a straightforward mapping back from a REG to NRAB DAGs that in turn represent executable Pig Latin expressions. Overall, the NRAB DAGs represented by REG are equivalent to the original NRAB DAGs that resulted in EG.

The choice of which alternative to pick for each equivalence node is guided by a *cost function*, the overall goal being to minimize the global cost of the plan. We assign a cost (weight) to each edge $n_1 \rightarrow n_2$ in the EG, representing all the processing cost (or effort) required to fully build the result of n_2 out of the result of n_1.

Figure 6 shows a possible REG produced for the EG depicted in Figure 5. This REG could have been for instance obtained by using a cost function based on counting the operator nodes in the optimized script. In the REG, each equivalence node has exactly one input edge, i.e., the *scans* and other operator nodes are shared across queries, whenever possible. In Section 5, we consider different cost functions and compare them in our experimental validation.

$$\text{Minimize: } \mathcal{C} = \sum_{e \in E} \mathcal{C}_e x_e$$

subject to:

$$x_e \in \{0,1\} \qquad \forall e \in E \qquad (1)$$

$$\sum_{e \in E_{t_o}^{in}} x_e = 1 \qquad \forall t_o \in T_o \qquad (2)$$

$$\sum_{e \in E_a^{in}} x_e = x_{E_a^{out}} \times |E_a^{in}| \qquad \forall a \in A \qquad (3)$$

$$\sum_{e \in E_o^{in}} x_e = \max_{e \in E_o^{out}} x_e \qquad \forall o \in O \qquad (4)$$

Figure 7: BIP reduction of the optimization problem.

3.3 Cost minimization based on binary integer programming

We model the problem of finding the minimum-cost REG relying on *Binary Integer Programming* (BIP), a well-explored branch of mathematical optimizations that has been used previously to solve many optimization problems in the database literature [16, 31]. Broadly speaking, a typical linear programming problem can be expressed as:

given a set of linear inequality constraints over a set of variables

find value assignments for the variables

such that the value of an objective function depending on these variables is minimized.

Such problems can be tackled by dedicated binary integer program *solvers*, some of which are by now extremely efficient, benefiting from many years of research and development efforts.

Generating the result equivalence graph. Given an input EG, for each of its nodes $n \in O \cup A \cup T_o$, we denote by E_n^{in} and E_n^{out} the sets of incoming and outgoing edges for n, respectively.

For each edge $e \in E$, we introduce a variable x_e, denoting whether or not e is part of the REG. Since in our specific problem formulation a variable x_e can only take values within $\{0,1\}$, our problem is formulated as a BIP problem.

Further, for each edge $e \in E$, we denote by $\mathcal{C}_e$ the cost $\mathcal{C}(e)$ assigned to e using some cost function $\mathcal{C}$. Importantly, the model we present in the following is independent of the chosen cost function.

Our optimization problem is stated in BIP terms in Figure 7. Equation (1) states that each x_e variable takes values in $\{0,1\}$. (2) ensures that every output is generated exactly once. (3) states that if the (only) outgoing edge of an operator node is selected, all of its inputs are selected as well. This is required in order for the algebraic operator to be capable of correctly computing its results. Finally, (4) states that if an equivalence node is generated, it should be used at least once, which is modeled by means of a *max* expression. Since *max* is not directly supported in the BIP model, the actual BIP constraints which we use to express (4) are:

ϵ	*map*	σ	π	*mapconcat*
■	□	□	□	□

$\uplus$	$\times$	*cogroup*	$\bowtie$	$\rtimes$	$\ltimes$	$\rtimes\!\!\ltimes$
■	□◇	□◇	□◇	□◇	□◇	□◇

□ *Child π operator can be swapped with the parent operator, iff none of the fields used by the parent operator is projected by π.*
◇ *Child π operator can be swapped with the parent operator only after rewriting the original π operator.*
■ *Child π operator cannot be swapped with the parent operator.*

Figure 8: Reordering and rewriting rules for π.

$$d_{e \in E_o^{out}} \in \{0,1\} \qquad \forall o \in O \qquad (4.1)$$

$$\sum_{e \in E_o^{in}} x_e \geq x_{e \in E_o^{out}} \qquad \forall o \in O \qquad (4.2)$$

$$\sum_{e \in E_o^{in}} x_e \leq (x_e - d_e + 1)_{e \in E_o^{out}} \qquad \forall o \in O \qquad (4.3)$$

$$\sum_{e \in E_o^{out}} d_e = 1 \qquad \forall o \in O \qquad (4.4)$$

These constraints encode the *max* constraint as follows. Equation (4.1) introduces a binary variable $d_{e \in E_o^{out}}$ used to model the *max* function. Equation (4.2) states that if an outgoing edge of an equivalence node is selected, then one of its incoming edges is selected too. (4.3) states that if no outgoing edge of an equivalence node is selected, then none of its incoming edges is selected. Further, (4.3) and (4.4) together ensure that if an outgoing edge of an equivalence node is selected, only one of its incoming edges will be selected. Observe that we can model the *max* function in this fashion since we know its value is bounded in $[0,1]$.

4. EFFECTIVE REUSE-BASED OPTIMIZATION

In this section, we introduce a set of techniques for identifying and exploiting additional subexpression factorization opportunities that go beyond those that are possible with the standard NRAB operators. The three extensions we bring to the basic PigReuse algorithm are: normalization (Section 4.1), join decomposition (Section 4.2), and aggressive merge (Section 4.3).

4.1 Normalization

Normalization of the input NRAB DAGs is carried out by *reordering* π operator nodes as follows: we push them away from *scan* operators or closer to *store* operators. We do this by visiting all operator nodes in a NRAB DAG, starting from a *scan*, and by moving each π operator up one level at a time. As we will shortly illustrate, *pushing projections up increases the chances to find equivalent subexpressions.*

Figure 8 spells out the conditions under which a π can be swapped with its parent operator. Each column in the topmost row represents a parent operator with which the child π may be swapped, and the value of each cell represents different conditions under which the swap is possible. For example, a child π can be swapped with a parent σ, iff the selection predicate does not carry over the attributes projected in π.

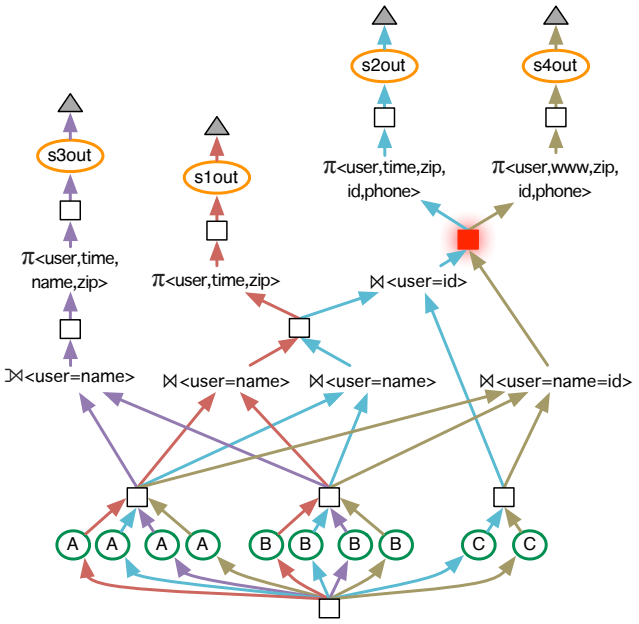


Figure 9: EG generated by PigReuse on the *normalized* NRAB DAGs q_1-q_4.

A special case is the *cogroup* operator. Since *cogroup* nests the input relations, reordering π with this operator requires complex rewriting. In particular, we will rewrite it into a *map* that applies the projection π on the bag of tuples corresponding to the input relation. *map* operators containing only combinations of *map* and π can still be pushed up following the conditions in Figure 8. This means that, in general, during normalization, one may need to introduce *map* operators nested more than two levels deep. Although the Pig Latin query language does not allow more than two levels of nested `FOREACH` expressions, our NRAB representation *map* allows it; furthermore, as we have found examining the code for executable plans within the Pig Latin engine, more than two levels of nesting *are* supported at the level of the execution engine[2].

Observe that operators such as ϵ or $\uplus$ restrict the possibilities of moving π operators across the DAG. It turns out also that they do not commute with the other algebraic operators; we term these "unmovable" operators, *bordering* operators in the sense that they raise borders to the moving of π across the DAG.

After our reuse-based algorithm produces the optimized REG, to avoid the performance loss incurred by manipulating many attributes at all levels (due to the pulling up of the projections), we push the π operators back, as close to the *scan* as possible. As our normalization algorithm may rewrite π operators using *map*, we extended the Pig Latin optimizer to support the (unnesting) rewriting of such cases, so that the π can be pushed back down through the plan. Recall that even if they cannot be pushed back down, the resulting plan (no matter how many levels the π operators are nested) will be executable by the Pig engine.

[2]The class `pig.newplan.logical.relational.LOForEach`, representing the `FOREACH` operator, has a field called `innerPlan` which in our tests could contain another `LOForEach` and so on on several levels. The purpose of introducing the language-level restriction may have been to prevent programmers from writing deeply-nested loops whose performance could be poor.

$$\frac{A_1 := cogroup\langle a_1, a_2, \ldots, a_n\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2, \ldots, \underline{\text{var}}_n) \quad A_2 := map\langle \delta(\underline{\text{var}}_1) \times \delta(\underline{\text{var}}_2) \times \ldots \times \delta(\underline{\text{var}}_n)\rangle(A_1)}{\bowtie \langle a_1, a_2, \ldots, a_n\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2, \ldots, \underline{\text{var}}_n) = A_2} \quad \text{(IJ)}$$

$$\frac{A_1 := cogroup\langle a_1, a_2\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2) \quad A_2 := map\langle \delta(\underline{\text{var}}_1) \times \delta(empty(\underline{\text{var}}_2)?\{\{\bot\}\} : \underline{\text{var}}_2)\rangle(A_1)}{⟕ \langle a_1, a_2\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2) = A_2} \quad \text{(LOJ)}$$

$$\frac{A_1 := cogroup\langle a_1, a_2\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2) \quad A_2 := map\langle \delta(empty(\underline{\text{var}}_1)?\{\{\bot\}\} : \underline{\text{var}}_1) \times \delta(\underline{\text{var}}_2)\rangle(A_1)}{⟖ \langle a_1, a_2\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2) = A_2} \quad \text{(ROJ)}$$

$$\frac{A_1 := cogroup\langle a_1, a_2\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2) \quad A_2 := map\langle \delta(empty(\underline{\text{var}}_1)?\{\{\bot\}\} : \underline{\text{var}}_1) \times \delta(empty(\underline{\text{var}}_2)?\{\{\bot\}\} : \underline{\text{var}}_2)\rangle(A_1)}{⟗ \langle a_1, a_2\rangle(\underline{\text{var}}_1, \underline{\text{var}}_2) = A_2} \quad \text{(FOJ)}$$

Figure 10: Decomposing JOIN operators.

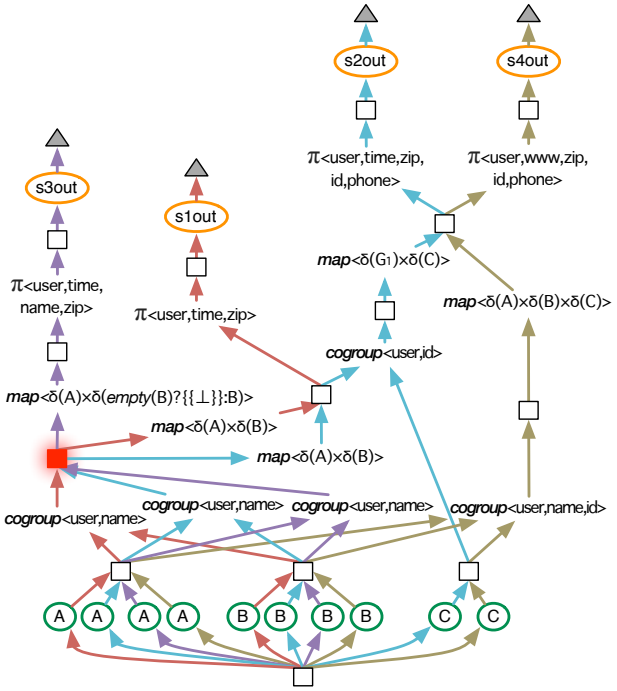


Figure 11: EG generated by PigReuse on the *normalized* and *decomposed* NRAB DAGs q_1-q_4.

To illustrate the advantages of our normalization phase, Figure 9 shows the EG generated by PigReuse over the *normalized* NRAB DAGs q_1 to q_4. Comparing this EG with the one shown in Figure 5, we see that due to the swapping of the π operator corresponding to q_2, our algorithm can identify an additional common subexpression between q_2 and q_4, by determining the equivalence between the joins over A, B, and C; the corresponding equivalence node is highlighted in Figure 9.

4.2 Join decomposition

The semantics of Pig Latin 's join operators e.g., $\bowtie$, ⟕, ⟖, or ⟗ allow rewriting (or *decomposing*) these operators into combinations of *cogroup* and *map* operators. The advantage of decomposing the joins in this way is that the result of the

$$\frac{\begin{array}{c}A_1 := cogroup\langle a'_1, a'_2, \ldots, a'_n\rangle(\underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \ldots, \underline{\mathsf{var}}'_n) \\ A_2 := \pi\langle group, \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_k\rangle(A_1) \qquad A_3 := \sigma\langle\neg(empty(\underline{\mathsf{var}}_1) \wedge \ldots \wedge empty(\underline{\mathsf{var}}_k))\rangle(A_2)\end{array}}{cogroup\langle a_1, \ldots, a_k\rangle(\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_k) = A_3 \mid \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_k \subset \underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \ldots, \underline{\mathsf{var}}'_n \wedge a_1, \ldots, a_k \subset a'_1, a'_2, \ldots, a'_n} \quad \text{(CG-CG)}$$

$$\frac{A_1 := cogroup\langle a'_1, a'_2, a'_3, \ldots, a'_n\rangle(\underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n) \qquad A_2 := map\langle\delta(\underline{\mathsf{var}}_1) \times \delta(\underline{\mathsf{var}}_2) \ldots \times \delta(\underline{\mathsf{var}}_k)\rangle(A_1)}{\bowtie\langle a_1, a_2, \ldots, a_k\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2, \ldots, \underline{\mathsf{var}}_k) = A_2 \mid \underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2, \ldots, \underline{\mathsf{var}}_k \subset \underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n \wedge a_1, a_2, \ldots, a_k \subset a'_1, a'_2, a'_3, \ldots, a'_n} \quad \text{(IJ-CG)}$$

$$\frac{A_1 := cogroup\langle a'_1, a'_2, a'_3, \ldots, a'_n\rangle(\underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n) \qquad A_2 := map\langle\delta(\underline{\mathsf{var}}_1) \times \delta(empty(\underline{\mathsf{var}}_2)?\{\{\bot\}\} : \underline{\mathsf{var}}_2)\rangle(A_1)}{⟕\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) = A_2 \mid \underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2 \subset \underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n \wedge a_1, a_2 \subset a'_1, a'_2, a'_3, \ldots, a'_n} \quad \text{(LOJ-CG)}$$

$$\frac{A_1 := cogroup\langle a'_1, a'_2, a'_3, \ldots, a'_n\rangle(\underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n) \qquad A_2 := map\langle\delta(empty(\underline{\mathsf{var}}_1)?\{\{\bot\}\} : \underline{\mathsf{var}}_1) \times \delta(\underline{\mathsf{var}}_2)\rangle(A_1)}{⟖\langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2) = A_2 \mid \underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2 \subset \underline{\mathsf{var}}'_1, \underline{\mathsf{var}}'_2, \underline{\mathsf{var}}'_3, \ldots, \underline{\mathsf{var}}'_n \wedge a_1, a_2 \subset a'_1, a'_2, a'_3, \ldots, a'_n} \quad \text{(ROJ-CG)}$$

Figure 12: Rules for aggressive merge.

cogroup operation, which does the heavy-lifting of assembling groups of tuples from which the *map* will then build join results, can be shared across different kinds of joins. The *map* will be different in each case depending on the join type, but the most expensive component of computing the join, namely the *cogroup*, will be factorized. Further, there is no noticeable performance difference between executing a certain join or its decomposed rewritten version.

Figure 10 shows the decomposition rules that are applied on the input NRAB DAGs.

Rule (IJ) rewrites an inner equi-join $\bowtie$ into two operators. The first one is a *cogroup* on the attributes used by the join predicate. The second one is a *map* that does the following for each input tuple: (*i*) project each bag of tuples corresponding to the *cogroup* input relations; (*ii*) apply a δ operation on each of those bags; and (*iii*) perform a cartesian product among the tuples resulting from unnesting those bags. Observe that if a bag is empty, e.g., the input relation did not contain any value for the given grouping value, the δ operator does not produce any tuple, and thus the tuples from the other bags for the given tuple are discarded. Thus, this rewriting produces the exact same result as the original $\bowtie$ operator.

The rest of the rules use the aggregation function *empty*, that checks if an input bag is empty. For instance, the expression $empty(\underline{\mathsf{var}})?\{\{\bot\}\} : \underline{\mathsf{var}}$ is a conditional assignment, that is: if $\underline{\mathsf{var}}$ is empty, a bag with a null tuple ($\bot$), i.e., a tuple whose values are bound to null values, conforming to $\underline{\mathsf{var}}$ schema is assigned, otherwise the bag $\underline{\mathsf{var}}$ is assigned.

Rule (LOJ) rewrites a left outer join ⟕ into a *cogroup* on the attributes used by the join predicate, followed by a *map* operator that (*i*) unnests the bag associated to the left input of the *cogroup*; (*ii*) if the bag associated to the right input ($plalias_2$) is empty, it replaces it with a bag with a null tuple, otherwise it keeps the bag as it is; (*iii*) unnests the bag resulting from the previous operation; and (*iv*) performs a cartesian product on the tuples resulting from the δ operations in order to generate the ⟕ result.

Rule (ROJ) rewrites a right outer join ⟖ in a similar fashion.

Finally, rule (FOJ) rewrites a full outer join ⟗ following the same principle as for the two previous operators. The difference is that in (FOJ) we check the bags from both inputs by means of the *empty* function.

Figure 11 shows the EG generated by PigReuse after applying *normalization and decomposition* to the NRAB DAGs

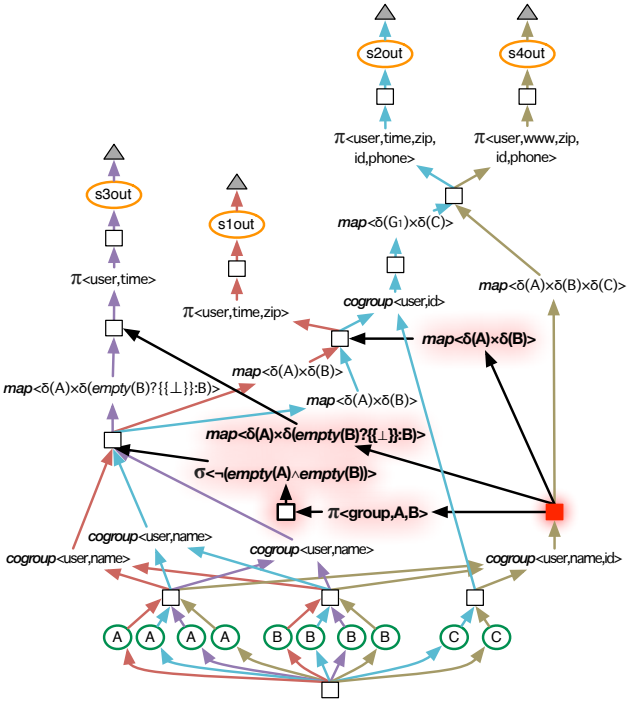


Figure 13: EG generated by PigReuse applying *aggressive merge* on the *normalized* and *decomposed* NRAB DAGs q_1-q_4.

q_1 to q_4. One can observe that the decomposition of the $\bowtie$ operators from q_1 and q_2, and the ⟕ operator from q_3 leads to an additional sharing opportunity, as the result of the *cogroup* on attributes *user* and *name* can be shared by the subsequent *map* operations (highlighted equivalence node).

4.3 Aggressive merge

The last extension we propose is based on the observation that it is possible to derive the results of a $\bowtie$ or *cogroup* operator from the results of a *cogroup*′ operator, as long as the former relies on *a subset* of the input relations and attributes of *cogroup*′. This means that these rewritings rely on the notion of *cogroup containment*. In particular, this entails checking the containment relationship between respective sets of input relations and attributes. Then, in order to generate the result of the original $\bowtie$ or *cogroup* operator, we

add the appropriate operator on top of *cogroup′*; this can be seen as a limited instance of query rewriting using views, where *cogroup′* plays the role of a view. In contrast to the previous extensions that are applied on the input NRAB DAGs, *aggressive merge* is applied while we are creating the EG.

Figure 12 shows the rewritings considered by our aggressive merge algorithm.

Rule (CG-CG) states that if a query contains a *cogroup′* operator with two or more input relations, any other *cogroup* (with at least one input relation, part of the *cogroup′* input) can be derived from the previous one in the following fashion. First, a π operator projects the subset of attributes that are needed for the result of the *cogroup* operator. Then, a σ operator discards the tuples where *all* the bags associated to each input relation are empty.

Rules (IJ-CG), (LOJ-CG), and (ROJ-CG) are similar to those shown in Figure 10; the only difference is that the *map* operators take only a subset of the bag attributes in the original *cogroup*. Note that we do not have a rule for the $\rtimes$ operator since we are able to generate its output directly from the result of the *cogroup*.

Figure 13 depicts the EG produced by PigReuse using the aggressive merge extensions, when normalization and decomposition has been applied to the NRAB plans q_1-q_4. The new connections created by aggressive merge are highlighted. The figure shows how the results for the *cogroup*, $\bowtie$, and $\rtimes$ operators on A and B relations are derived from the *cogroup* operator on A, B, and C.

5. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We have implemented PigReuse, our reuse-based optimization approach, in Java 1.6. The source code amounts to about 8000 lines and 50 classes. It works on top of Apache Pig 0.12.1 [22], which relied on the Hadoop platform 1.1.2 [12]. The cost-based plan selection algorithm (Section 3.2) uses the Gurobi BIP solver 5.6.2 [11].

Section 5.1 describes our experimental setup. Then, Section 5.2 presents the two alternative cost functions that we have implemented and experimented with; recall that while the cost function does impact the configuration chosen by the BIP solver, our approach and algorithms are independent of the cost function chosen. Finally, Section 5.3 presents our experimental results.

5.1 Experimental setup

Deployment. All our experiments run in a cluster of 8 nodes connected by a 1GB Ethernet. Each node has a 2.93GHz Quad Core Xeon processor and 16GB RAM. The nodes run Linux CentOS 6.4. Each node has two 600GB SATA hard disks where HDFS is mounted.

Setup. In our experiments, we use the PigLatin scripts part of the PigMix [23] performance benchmark.

We have created the dataset using the PigMix generator. In particular, we created a `page_views` input file with 12.5 million rows; other input files are based on this one, but they are much smaller. The data set amounted to approximately 20 GB before the 3-way replication applied by HDFS.

We run our algorithm with two different workloads. The first one (W_1) consisted on a subset of 12 scripts provided

Figure 14: PigReuse evaluation using workload W_1 (left) and W_2 (right).

by the PigMix benchmark (l_2-l_7 and l_{11}-l_{16}), that contain operators that are supported currently in our implementation such as `JOIN`, `COGROUP`, `FILTER`, etc. Each script has on average 7 operators; more details about them can be found in [23]. The second workload (W_2) consisted of the scripts of W_1, as well as 8 additional scripts that feature many `JOIN` flavours, `COGROUP` on many relations, etc. This second collection of scripts was created to validate our algorithms more extensively, based on the same data.

5.2 Cost functions

We now present the two cost functions that are implemented currently in PigReuse, which focus on reducing the number of operations and number MapReduce jobs in the final plan, respectively. Although more elaborated cost functions can be envisioned, these two already lead to considerable execution time gains, as our experiments shortly show.

Eliminating operators and/or MapReduce jobs as a consequence of our *reuse-based optimization* leads to a reduction in the total work entailed by the execution of the input Pig Latin batch of scripts, and thus a decrease of the response time. In particular, the reduction in the number of MapReduce jobs is very significant for the response time of the script batch, since each job has a significant constant setup time related to reading from and writing to files, launching the slave nodes etc.

Number of operators. A first metric capturing the effort required by the evaluation of a batch of Pig Latin scripts is the number of operators in the equivalent NRAB expression eventually evaluated, that is :

$$\begin{array}{ll} \mathcal{C}_e = 1 & \forall e \in E_a^{out},\ \forall a \in A \\ \mathcal{C}_e = 0 & \text{for all the rest} \end{array}$$

Above, we assign a cost of 1 to the execution of every algebraic operator a, and we attach this cost to its outgoing edge. All the other edges, i.e., incoming edges to an operator node, have a cost of 0.

Number of MapReduce jobs. Our second cost function is closely related to the Pig execution engine on top of MapReduce. The function minimizes the MapReduce jobs needed to compute the results of the input Pig Latin scripts, as some groups of operators are executed by Pig as part of the same job. For instance, σ, π, and *map* do not generate a new MapReduce job, which is very convenient for our *decomposition* and *aggressive merge* extension techniques that introduce these operators quite aggressively when rewriting.

5.3 Experimental results

We now study the benefits brought by the optimizations proposed in this work. The reported results are averaged over three runs.

Figure 14 shows the effectiveness of our baseline PigReuse algorithm (PR), PigReuse with *normalization* (PR+N), PigReuse with *normalization* and *decomposition* (PR+ND),

	PigReuse			
	PR	**PR+N**	**PR+ND**	**PR+NDA**
W_1 - EG equivalent nodes (#)	58	59	60	62
W_1 - EG operator nodes (#)	83	79	83	87
W_1 - REG (*minop*) operator nodes (#)	57	58	59	59
W_1 - REG (*minmr*) operator nodes (#)	57	58	59	60
W_2 - EG equivalent nodes (#)	74	82	83	88
W_2 - EG operator nodes (#)	135	125	131	143
W_2 - REG (*minop*) operator nodes (#)	73	81	82	82
W_2 - REG (*minmr*) operator nodes (#)	73	81	82	85

Table 2: Reuse-based optimization details for workloads W_1 and W_2.

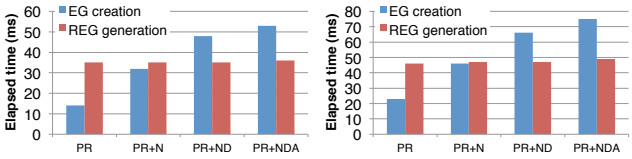


Figure 15: PigReuse compile time overhead for workloads W_1 (left) and W_2 (right).

and PigReuse applying all our extensions including *aggressive merge* (PR+NDA). The cost function that minimizes the total number of operators in the EG is denoted by *minop*, while the cost function that minimizes the total number of MapReduce jobs is denoted by *minmr*.

First, one can notice that our PigReuse algorithms reduce the total execution time by more than 60% on average.

For the workloads we considered, *normalization* did not play a major role reducing the execution time over the baseline PigReuse algorithm. However, once the join operators were *decomposed*, the execution time improved slightly over PR and PR+N due to the reutilization of *cogroup* results among multiple joins.

When *aggressive merge* was applied, the execution time decreased only if the *minmr* cost function was used. The reason is that if the *minop* function is used, PigReuse generates the same REG for PR+ND and PR+NDA, namely, the REG with the minimum number of operators. However, if the *minmr* cost function is used, PigReuse chooses an alternative plan that executes faster even if though it consists of more operators.

Table 2 shows details about the EGs and REGs created by PigReuse. The PigReuse algorithm reduces the total number of logical operators by an average of 30%, using any of the given cost functions. In particular, the REG generated by PigReuse using the *minop* or *minmr* cost functions has the same number of operators, except when *aggressive merge* is used (PR+NDA). The reason is that all the connections that we establish through the aggressive merge strategy do not result in extra MapReduce jobs. Thus, using that strategy and the *minmr* cost function, a plan that contains more nodes but translates into less MapReduce jobs is selected. As we have seen before, this alternative plan leads to considerable execution time savings.

Finally, Figure 15 shows the total compile time overhead of using PigReuse. EG creation time includes the time to generate the EG, i.e., identifying equivalent expressions and merging them, and the time to apply our extensions to the algorithm (if any). We can observe that the EG creation time increases as the extensions to the baseline PigReuse are applied. On the other hand, the time to generate the REG is almost constant among all the strategies. It is important to note that the total optimization time stays below 125 ms in all cases. This means that with a relatively small overhead, PigReuse obtains a very considerable execution time improvement.

6. RELATED WORKS

Our work relates to several areas of existing research.

Relational multi-query optimization. Early works on multi-query optimization (MQO) [15, 27] sought to improve the performance of query batches featuring common subexpressions, thus they are the most directly related to our work. These works proposed exhausive, expensive optimization algorithms which were not integrated with existing system optimizers. [26] was the first to integrate MQO into a Volcano-style optimizer, while [32] presents a completely integrated MQO solution also comprising the maintenance and exploitation of materialized views. Finally, the recent [28] presents a MQO approach taking into account the physical requirements (e.g., data partitioning) of the consumers of common sub-expressions in order to propose globally optimal execution plans.

To the best of our knowledge, equivalence or containment-based optimizations on the NRAB representation of Pig Latin scripts has not been studied before. We argue that our formalization into NRAB (which we are the first to provide) lays the adequate foundation for our reuse-based optimization, with correctness guarantees.

Reuse-based optimizations on MapReduce. Multiple works have focused on avoiding redundant processing for a batch of MapReduce jobs by sharing their (intermediate) results [1, 7, 20, 30]. In contrast to the PigReuse approach, these works either (*i*) need some information about the MapReduce job semantics in order to be efficient [7], or (*ii*) their detected reuse-based optimization opportunities are limited to inputs and outputs of the mappers and reducers [1, 20, 30]. Our PigReuse algorithm works on the semantic representation of Pig Latin scripts which enable complex reuse-based optimizations, e.g., based on rewritings of expressions, and then it connects the NRAB representation to the real execution effort through a customizable cost function. The very recent works [17, 18] consider reusing results stored by MapReduce in the distributed file system for failure resilience reasons as materialized views, based on which subsequent queries can be evaluated faster. While the general idea of reusing results is the same, the language they consider is HiveQL, and these works do not focus on providing a complete and provably correct reuse approach.

Single query optimization for MapReduce jobs. Recent works have proposed optimizations for MapReduce jobs [13, 14]. Our approach is orthogonal and complimentary to these optimizations, as we can detect common subexpressions among batches of Pig Latin queries at the higher level, and then these optimizations may be applied on the MapReduce jobs generated by the Pig engine.

Optimization using integer programming. Integer programming has been used before to model different optimization problems in data management systems, e.g., in materialized view selection and maintenance [31], or optimal utilization of materialized views in publish/subscribe systems [16]. Although our optimization goal is different, we got inspiration from these works to model the cost-based plan selection using integer programming.

7. CONCLUSION

We have presented a novel approach for identifying and reusing common subexpressions occurring in Pig Latin scripts. In particular, we lay the foundation of our reuse-based algorithms by formalizing the semantics of the Pig Latin query language with Extended Nested Relational Algebra for Bags. Our PigReuse algorithm identifies subexpression merging opportunities, and selects the best ones to merge based on a cost-based search process implemented with the help of a linear program solver. The output of our algorithm is a merged script reducing a given cost function. Our experimental results demonstrate the value of our reuse-based algorithms and optimization strategies.

8. REFERENCES

[1] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *PVLDB*, 2008.

[2] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB*, 2011.

[3] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *ICDT*. 1990.

[4] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 2011.

[5] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *PODS*, 1993.

[6] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical report, 1994.

[7] I. Elghandour and A. Aboulnaga. ReStore: reusing results of MapReduce jobs. *PVLDB*, 2012.

[8] G. Graefe. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[9] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. *SIGMOD Record*, 1995.

[10] S. Grumbach and T. Milo. Towards Tractable Algebras for Bags. In *PODS*, 1993.

[11] Gurobi Optimizer. `http://www.gurobi.com`.

[12] Apache Hadoop. `http://hadoop.apache.org/`.

[13] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *PVLDB*, 2011.

[14] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. *PVLDB*, 2011.

[15] M. Jarke. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems*, pages 191–205. Springer, 1985.

[16] K. Karanasos, A. Katsifodimos, and I. Manolescu. Delta: Scalable data dissemination under capacity constraints. *PVLDB*, 2013.

[17] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: Souping Up Big Data Query Processing with a Multistore System. In *SIGMOD*, 2014.

[18] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic Physical Design for Big Data Analytics. In *SIGMOD*, 2014.

[19] L. Libkin and L. Wong. Query Languages for Bags and Aggregate Functions. *Journal of Computer and System Sciences*, 1997.

[20] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: sharing across multiple queries in MapReduce. *PVLDB*, 2010.

[21] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[22] Apache Pig. `http://pig.apache.org/`.

[23] PigMix. `http://cwiki.apache.org/confluence/display/PIG/PigMix`.

[24] A. Poulovassilis and C. Small. Algebraic query optimisation for database programming languages. *The VLDB Journal*, 1996.

[25] C. Re, J. Siméon, and M. Fernandez. A complete and efficient algebraic compiler for XQuery. In *ICDE*, 2006.

[26] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 249–260. ACM, 2000.

[27] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[28] Y. N. Silva, P.-A. Larson, and J. Zhou. Exploiting Common Subexpressions for Cloud Query Processing. In *ICDE*, 2012.

[29] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a PB scale data warehouse using Hadoop. In *ICDE*, 2010.

[30] G. Wang and C.-Y. Chan. Multi-Query Optimization in MapReduce Framework. *PVLDB*, 2013.

[31] J. Yang. Algorithms for materialized view design in data warehousing environment. *PVLDB*, 1997.

[32] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.

APPENDIX

A. EXTENDED NESTED RELATIONAL ALGEBRA FOR BAGS

First, we recall the NRAB [10] data model in Section A.1, while we present the subset of its operators that we use to represent Pig Latin semantics in Section A.2. Then, Section A.3 extends NRAB with the Pig Latin operators, whose semantics are defined using the subset of NRAB operators that we introduce previously.

A.1 Data model

Let us assume the existence of a set of domain names $\widehat{D}_1, \ldots, \widehat{D}_n$ and an infinitive set of attributes $a_1, a_2, \ldots$. Further, the domain names are associated with domains $D_1, \ldots, D_n$. The elements of the domains can be of either atomic type or complex type. A type is associated with each instance of a domain. Formally, *types* and *values* are defined as follows:

- If $\widehat{D}_i \in \widehat{D}$ is a domain name, then $\widehat{D}_i$ denotes the *domain type*. For each database relation R in domain $\widehat{D}$, the type of R is $\widehat{D}_i$.
- If $T_1, \ldots, T_n$ are types and $a_1, \ldots, a_n$ are distinct attribute names for tuples in a database relation R, then R=$\{\{[a_1 : T_1, ..., a_n : T_n]\}\}$ is a bag of tuples in which $[a_1 : T_1, ..., a_n : T_n]$ is a *tuple* type. If v_1, ..., v_n are values of types T_1, ..., T_n, respectively, then $[a_1 : v_1, ..., a_n : v_n]$ is value of the *tuple* type. We also include $T_{[]}$ as a type; the only value of this type is [], the empty tuple.
- A *bag* is a (homogeneous) collection of tuples that may contain duplicates. If T is a tuple type, then $\{\{T\}\}$ is a *bag* type, whose domain is a set of bags containing homogeneous tuples of type T. We say that an element o *n-belongs* to a bag, if element o has n occurrences in that bag.
- A *bag database* is a set of named bags. A *bag schema* is an expression $B : T$, where B is a bag name and T is a bag type. An instance of B is a bag of type T.

A.2 Basic operators

NRAB operators. We now describe the NRAB operators [10] that we use to express Pig Latin sementics. The input and output types of all these operators are *bag* type.

- Duplicate elimination (ϵ). This operator extracts the distinct tuples in a relation. $\epsilon(R)$ is a bag containing exactly one occurrence of each tuple in R i.e., an element o *1-belongs* to $\epsilon(R)$ iff o *p-belongs* to R for some $p > 0$, and *0-belongs* to $\epsilon(R)$ otherwise.
- Restructuring (*map*). $map\langle\varphi\rangle(R)$ returns a bag of type $\{\{T\}\}$, constructed by applying a function φ on each element of R. This operation is introduced for performing restructuring of complex values, which may include the application of functions to substructures of the values. *map* is a higher order operation with a function parameter φ that describes the restructuring.
- Selection (σ). Given a bag R and a boolean valued predicate condition p, $\sigma\langle p\rangle(R)$ denotes the select operation that returns a bag containing all the elements of R that satisfy the condition p. Only unary predicates can be used as parameters for the select; we refer to them as *select specifications*.
- Additive union ($\uplus$). This operator deals with the union of bags with possibly duplicate elements. If R and S are two input relations of *bag* type $\{\{T\}\}$, then $R \uplus S$ is a bag of type $\{\{T\}\}$, such that a tuple t of type T *n-belongs* to $R \uplus S$, iff t *p-belongs* to R and *q-belongs* to S and $n = p + q$.
- Substraction ($-$). If R and S are two input relations of *bag* type $\{\{T\}\}$, then $R-S$ is a bag of type $\{\{T\}\}$, such that a tuple t of type T *n-belongs* to $R-S$, iff t *p-belongs* to R, *q-belongs* to S and $n = max(0, p - q)$, where function *max* returns the highest among the input values 0 and $p - q$.
- Cartesian product ($\times$). If R and S are bags containing tuples of arity k and k' respectively, then $R \times S$ is a bag containing tuples of arity $k + k'$, such that the new relation X becomes, $X = R \times S = \{[a_1, \ldots, a_k, a_{k+1}, \ldots, a_{k+k'}]\}$, where $[a_1, \ldots, a_k, a_{k+1}, \ldots, a_{k+k'}]$ is a tuple type. Tuple t $=[a_1, \ldots, a_k, a_{k+1}, \ldots, a_{k+k'}]$ *n-belongs* to $R \times S$ iff $t_1 = [a_1, \ldots, a_k]$ *p-belongs* to R and $t_2 = [a_{k+1}, \ldots, a_{k+k}]$ *q-belongs* to S and $n = pq$.
- Bag-destroy function (δ). δ unnests one level of bag nesting. If R is a bag of type $\{\{S : \{\{T\}\}\}\}$, then $map\langle\delta(S)\rangle(R)$ results a bag of type {{T}}.

NRAB functions. Function definition in NRAB has two parts: a class of base functions and function constructors that are used for constructing more complex function expressions.

First, we describe the base functions. In our algebra, constants c, and database relation names $\widehat{R}$ are considered as functions. Additionally, each attribute of the input relation is also considered as a function expression. We use **id** for denoting the identity function. For example, $map\langle R \uplus \mathbf{id}\rangle(S)$, denotes that additive union of R's element is performed recursively on each of S's elements, where S is a bag of tuples. Here, **id** indicates each element in S. The algebraic operations, except *select* and *restructuring*, are function expressions. *Select* and *restructuring* are function constructors, which are discussed next.

In our algebra, complex functions are constructed by using one of the function construction operators (*select* and *restructuring*). If φ is a unary function, then $map\langle\varphi\rangle(R)$ is a function. Similarly, if p is a unary boolean-valued function then $\sigma\langle p\rangle(R)$ is also a function. We use tuple construction as a function constructor i.e., if $f_1, \ldots, f_n$ are unary functions, then $[f_1, \ldots, f_n]$ is a unary function, whose meaning is defined by $[f_1, \ldots, f_n](x) = [f_1(x), \ldots, f_n(x)]$. Our algebra supports labeled tuple construction as a function constructor too, i.e., formation of expressions like $[A_1 = f_1, \ldots, A_n = f_n]$ is allowed; note that the A_is here are not functions but labels. The semantics is given by $[A_1 = f_1, \ldots, A_n = f_n](x)$ $= [A_1 : f_1(x), \ldots, A_n : f_n(x)]$. This implies that every function is unary, where its input is a tuple.

A.3 Additional operators

In the following, we extend the basic NRAB set of operators to encapsulate the semantics of more complex operations that are supported by the Pig Latin language.

- Scan (*scan*). $scan\langle fileID\rangle$ is an operator introduced to represent a data source that reads a file *fileID*.
- Store (*store*). $store\langle dir\rangle(R)$ is an operator introduced to represent a data sink that writes the bag R to directory *dir*.
- Projection (π). $\pi\langle a_1,\ldots,a_n\rangle(R)$ projects attributes with names $a_1,\ldots,a_n$ from the tuples in bag R. Formally:

$$\pi\langle a_1,\ldots,a_n\rangle(R) \equiv map\langle[a_1,\ldots,a_n]\rangle(R)$$

- Cogroup (*cogroup*). In order to define the semantics of the *cogroup* operator, we first define a G operator that works on a single bag. In particular, $G\langle a\rangle(R)$ groups the tuples in R by the value bound to a. The result of the expression is a bag with tuples containing two elements: a *group* attribute associated to the grouping value, and a R attribute associated to the bag of tuples whose attribute a was bound to that value. Formally:

$$G\langle a\rangle(R) \equiv map\langle map\langle\sigma\langle group{=}a\rangle(id)\rangle(R)\rangle \\ (map\langle[group = a, R = R]\rangle(R))$$

$cogroup\langle a_1,\ldots,a_n\rangle(R_1,\ldots,R_n)$ groups together tuples from multiple bags $R_1,\ldots,R_n$, based on the values of their attributes $a_1,\ldots,a_n$, respectively. The result of a *cogroup* operation is a bag containing a *group* attribute, bound to values of attributes $a_1,\ldots,a_n$, followed by one bag of grouped tuples for each relation in $R_1,\ldots,R_n$. Without loss of generality, we define it formally for two input relations; the extension for more than two inputs is straightforward. Thus:

$$cogroup\langle a_1,a_2\rangle(R_1,R_2) \equiv A_9$$

where:

$$A_1 := G\langle a_1\rangle(R_1) \qquad A_2 := G\langle a_2\rangle(R_2)$$
$$A_3 := \bowtie\langle group{=}group\rangle(A_1,A_2) \quad A_4 := \pi\langle group\rangle(A_3)$$
$$A_5 := \pi\langle group\rangle(A_1)$$
$$A_6 := \bowtie\langle group{=}group\rangle(A_5{-}A_4,A_1)$$
$$A_7 := \pi\langle group\rangle(A_2)$$
$$A_8 := \bowtie\langle group{=}group\rangle(A_7{-}A_4,A_2)$$
$$A_9 := A_3 \uplus A_6 \uplus A_8$$

- Inner join ($\bowtie$). $\bowtie\ \langle a_1{=}\ldots{=}a_n\rangle(R_1,\ldots,R_n)$ creates the cartesian product between the tuples in bags $R_1,\ldots,R_n$, and filters the resulting tuples based on condition $a_1{=}\ldots{=}a_n$. Thus, $\bowtie$ is formalized as:

$$\bowtie\langle a_1{=}\ldots{=}a_n\rangle(R_1,\ldots,R_n) \equiv \\ \sigma\langle a_1{=}\ldots{=}a_n\rangle(R_1\times\ldots\times R_n)$$

- Left outer join ($\rtimes$). $\rtimes\ \langle a_1{=}a_2\rangle(R_1,R_2)$ returns the cartesian product of tuples from input relations R_1 and R_2 for which boolean condition $a_1{=}a_2$ is true, and the tuples in R_1 without a matching right tuple. Formally:

$$\rtimes\langle a_1,a_2\rangle(R_1,R_2) \equiv A_5$$

where:

$$A_1 := \bowtie\langle a_1{=}a_2\rangle(R_1,R_2) \qquad A_2 := \pi\langle a_1\rangle(A_1)$$
$$A_3 := \pi\langle a_1\rangle(R_1) \quad A_4 := \bowtie\langle a_1{=}a_1\rangle(A_3{-}A_2,A_1)$$
$$A_5 := A_1 \uplus A_4$$

- Right outer join ($\ltimes$). $\ltimes\ \langle a_1{=}a_2\rangle(R_1,R_2)$ returns the cartesian product of tuples from input relations R_1 and R_2 for which boolean condition $a_1{=}a_2$ is true, and the tuples in R_2 without a matching right tuple. Formally:

$$\ltimes\langle a_1,a_2\rangle(R_1,R_2) \equiv A_5$$

where:

$$A_1 := \bowtie\langle a_1{=}a_2\rangle(R_1,R_2) \qquad A_2 := \pi\langle a_2\rangle(A_1)$$
$$A_3 := \pi\langle a_2\rangle(R_2) \quad A_4 := \bowtie\langle a_2{=}a_2\rangle(A_3{-}A_2,A_1)$$
$$A_5 := A_1 \uplus A_4$$

- Full outer join ($\rtimes\!\ltimes$). $\rtimes\!\ltimes\ \langle a_1{=}a_2\rangle(R_1,R_2)$ returns the cartesian product of tuples from input relations R_1 and R_2 for which boolean condition $a_1{=}a_2$ is true, the tuples in R_1 without a matching right tuple, and the tuples in R_2 without a matching left tuple. Formally:

$$\rtimes\!\ltimes\langle a_1,a_2\rangle(R_1,R_2) \equiv A_8$$

where:

$$A_1 := \bowtie\langle a_1{=}a_2\rangle(R_1,R_2)$$
$$A_2 := \pi\langle a_1\rangle(A_1) \quad A_3 := \pi\langle a_1\rangle(R_1)$$
$$A_4 := \bowtie\langle a_1{=}a_1\rangle(A_3{-}A_2,A_1)$$
$$A_5 := \pi\langle a_2\rangle(A_1) \quad A_6 := \pi\langle a_2\rangle(R_2)$$
$$A_7 := \bowtie\langle a_2{=}a_2\rangle(A_6{-}A_5,A_1)$$
$$A_8 := A_1 \uplus A_4 \uplus A_7$$

- Restructuring and concatenation (*mapconcat*). The operation $mapconcat\langle\varphi\rangle(R)$ applies $map\langle\varphi\rangle(R)$ and concatenates its result to the original tuple. Thus:

$$mapconcat\langle\varphi\rangle(R) \equiv map\langle[id,\varphi]\rangle(R)$$

- Empty (*empty*) and aggregate functions (*aggr*). The boolean function $empty(R)$ returns true iff R is empty. In turn, aggregate functions *aggr* include *count*, *max*, *min* and *sum*. $count(R)$ calculates the number elements in a bag of tuples R. $max\langle a\rangle(R)$ returns the maximum integer value of an element a in a bag of tuples R. $min\langle a\rangle(R)$ returns the minimum integer value of an element a in a bag of tuples R. $sum\langle a\rangle(R)$ returns the sum of integer values for an element a in a bag of tuples R. Each of these functions can be described in NRAB. For the sake of presentation, we do not further describe the semantics of these functions in this work.

B. TRANSLATION RULES FOR PIG LATIN OPERATORS

Figure 16 shows the translation rules for the Pig Latin operators that have a one-to-one correspondence with NRAB operators. In the following we describe each of these rules.

$$\frac{A := scan\langle \mathit{fileID}\rangle}{\texttt{LOAD}\ \mathit{fileID} \Rightarrow A} \qquad \text{(Load)}$$

$$\frac{A := \epsilon(\underline{\mathsf{var}}_1)}{\texttt{DISTINCT}\ \underline{\mathsf{var}}_1 \Rightarrow A} \qquad \text{(Distinct)}$$

$$\frac{A := \sigma\langle boolexpr\rangle(\underline{\mathsf{var}}_1)}{\texttt{FILTER}\ \underline{\mathsf{var}}_1\ \texttt{BY}\ boolexpr \Rightarrow A} \qquad \text{(Filter)}$$

$$\frac{A := \delta(\underline{\mathsf{var}})}{\texttt{FLATTEN}(\underline{\mathsf{var}}) \Rightarrow A} \qquad \text{(Flatten Function)}$$

$$\frac{A := empty(\underline{\mathsf{var}})}{\texttt{IsEmpty}(\underline{\mathsf{var}}) \Rightarrow A} \qquad \text{(Empty Function)}$$

$$\frac{A := aggr(\underline{\mathsf{var}})}{\texttt{AGGR}(\underline{\mathsf{var}}) \Rightarrow A} \qquad \text{(Aggregation Function)}$$

$$\frac{A := \underline{\mathsf{var}}_1 \uplus \ldots \uplus \underline{\mathsf{var}}_n}{\texttt{UNION}\ \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n \Rightarrow A} \qquad \text{(Union)}$$

$$\frac{A := \underline{\mathsf{var}}_1 \times \ldots \times \underline{\mathsf{var}}_n}{\texttt{CROSS}\ \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n \Rightarrow A} \qquad \text{(Cross)}$$

$$\frac{A := cogroup\langle a_1, \ldots, a_n\rangle(\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n)}{\texttt{COGROUP}\ \underline{\mathsf{var}}_1\ \texttt{BY}\ a_1, \ldots, \underline{\mathsf{var}}_n\ \texttt{BY}\ a_n \Rightarrow A} \qquad \text{(Gogroup)}$$

$$\frac{A_1 := \bowtie \langle a_1, \ldots, a_n\rangle(\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n)}{\texttt{JOIN}\ \underline{\mathsf{var}}_1\ \texttt{BY}\ a_1, \ldots, \underline{\mathsf{var}}_n\ \texttt{BY}\ a_n \Rightarrow A_1} \qquad \text{(Inner Join)}$$

$$\frac{A_1 := \text{⟕} \langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2)}{\texttt{JOIN}\ \underline{\mathsf{var}}_1\ \texttt{BY}\ a_1\ \texttt{LEFT}, \underline{\mathsf{var}}_2\ \texttt{BY}\ a_2 \Rightarrow A_1} \qquad \text{(Left Outer Join)}$$

$$\frac{A_1 := \text{⟖} \langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2)}{\texttt{JOIN}\ \underline{\mathsf{var}}_1\ \texttt{BY}\ a_1\ \texttt{RIGHT}, \underline{\mathsf{var}}_2\ \texttt{BY}\ a_2 \Rightarrow A_1} \qquad \text{(Right Outer Join)}$$

$$\frac{A_1 := \text{⟗} \langle a_1, a_2\rangle(\underline{\mathsf{var}}_1, \underline{\mathsf{var}}_2)}{\texttt{JOIN}\ \underline{\mathsf{var}}_1\ \texttt{BY}\ a_1\ \texttt{FULL}, \underline{\mathsf{var}}_2\ \texttt{BY}\ a_2 \Rightarrow A_1} \qquad \text{(Full Outer Join)}$$

Figure 16: Rules for translating Pig Latin operators to corresponding NRAB representations.

Rule (Load) translates a `LOAD` expression into a *scan* that generating a new bag that satisfies the schema description in the input expression.

Rule (Distinct) translates `DISTINCT` into a ϵ operator on the input relation $\underline{\mathsf{var}}_1$.

Rule (Filter) translates a Pig Latin `FILTER` operator into a selection σ with a condition *boolexpr* on $\underline{\mathsf{var}}_1$.

Rule (Flatten Function) translates `FLATTEN` into a δ function that unnests the bag $\underline{\mathsf{var}}$.

Rule (Aggregation Function) translates Pig Latin aggregation functions into the NRAB aggregate operators counterparts.

The functions introduced in the last two rules are blocks that need to be used in the algebra in conjunction with an *map* operator.

Rule (Cross) translates a `CROSS` into a cartesian product between $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$.

Rule (Gogroup) translates a Pig Latin `COGROUP` operation to its algebraic equivalence *cogroup* that groups the tuples in $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$ based on the values of attributes bound to $a_1, \ldots, a_n$.

Rule (Inner Join) translates an inner join `JOIN` operator into its algebraic counterpart $\bowtie$. Rule (Left Outer Join) translates a Pig Latin left outer join expression into a ⟕ operator, while rule (Right Outer Join) translates a Pig Latin right outer join expression into a ⟖ operator. Finally, rule (Full Outer Join) translates a Pig Latin full outer join expression into a ⟗ operator. Observe that outer joins can only be binary in Pig Latin.